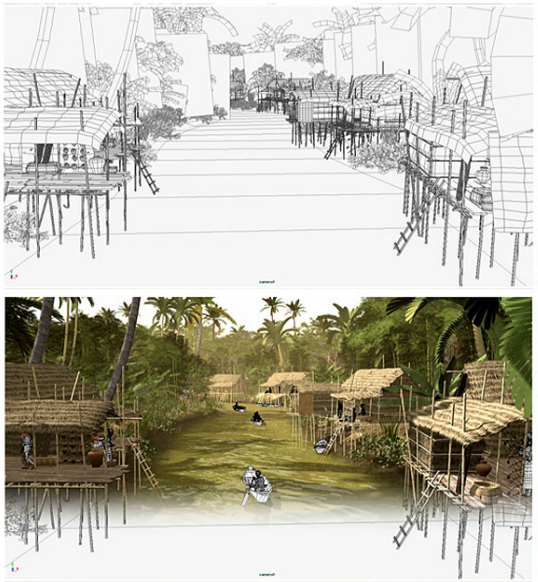
MONASH University

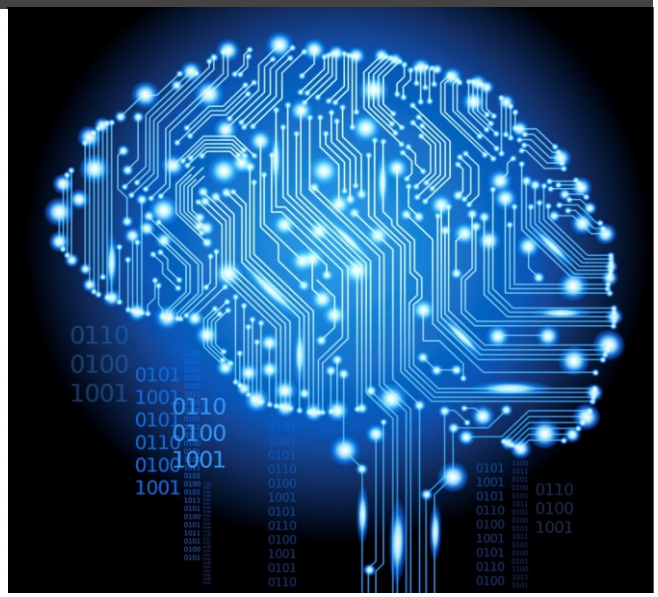
Information Technology

FIT1008/FIT2085 Lecture 8

Function calling

Prepared by: M. Garcia de la Banda
based on D. Albrecht, J. Garcia





Where are we at:

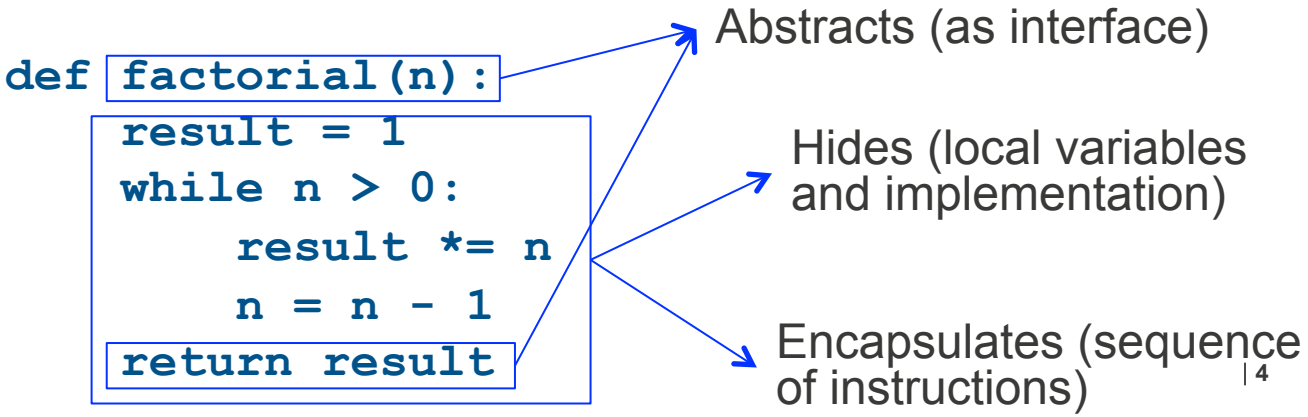
- **Discussed MIPS architecture and main design decisions**
- **Gone through the instruction set that we will use in this unit**
- **Discussed how variables are stored and accessed**
  - Global variables in the data segment
  - Local variables in the stack segment (the system stack)
    - Memory diagrams
    - Allocation/deallocation with `$sp`
    - Access by a negative offset from `$fp`
- **Practiced translation of decisions**
  - selection (if) and
  - loops (while and for)
- **Discussed how to create and access arrays of integers**

Learning objectives for this lecture

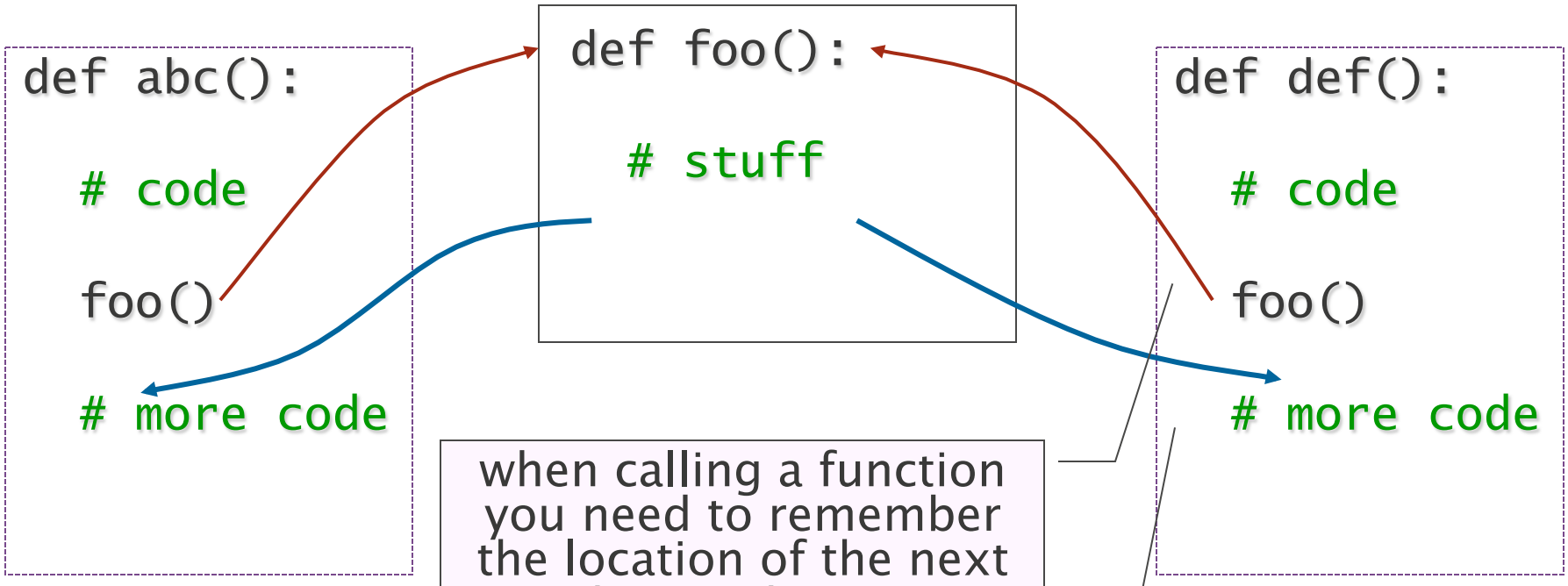
- **To understand how functions are implemented in MIPS**
- **In particular:**
  - Use of the `jal` and `jr` instructions
  - Use of the system stack to satisfy function properties
- **To understand the reasons behind the decisions taken by the function calling convention**
- **To understand what a stack frame is, and its purpose**
- **To be able to implement a function call in MIPS**

Reminder: why using functions?

- **As encapsulation of a sequence of instructions:**
  - Are self-contained
  - Can have their own private (local) variables/data
  - Can be called repeatedly (reuse)
- **As abstractions:**
  - Can be generalized by taking parameters
  - Can inform through return values
- **As hiders of information**
  - Make sure caller cannot access/modify internal data



Function calling: return where?

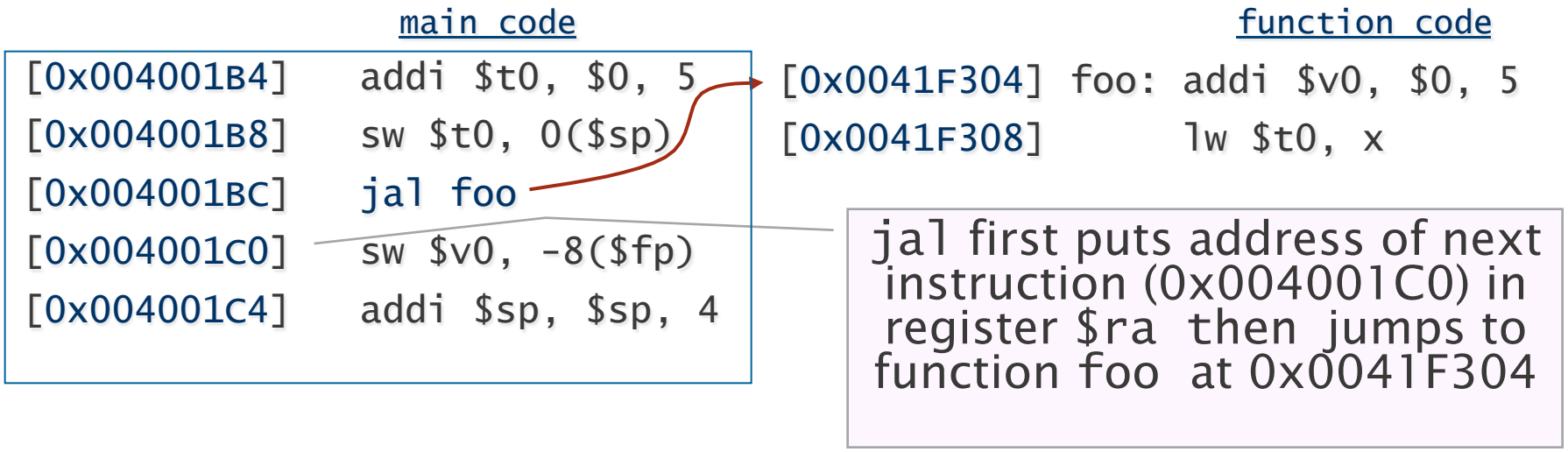


when calling a function you need to remember the location of the next instruction ...

... so you can return to it when the function finishes

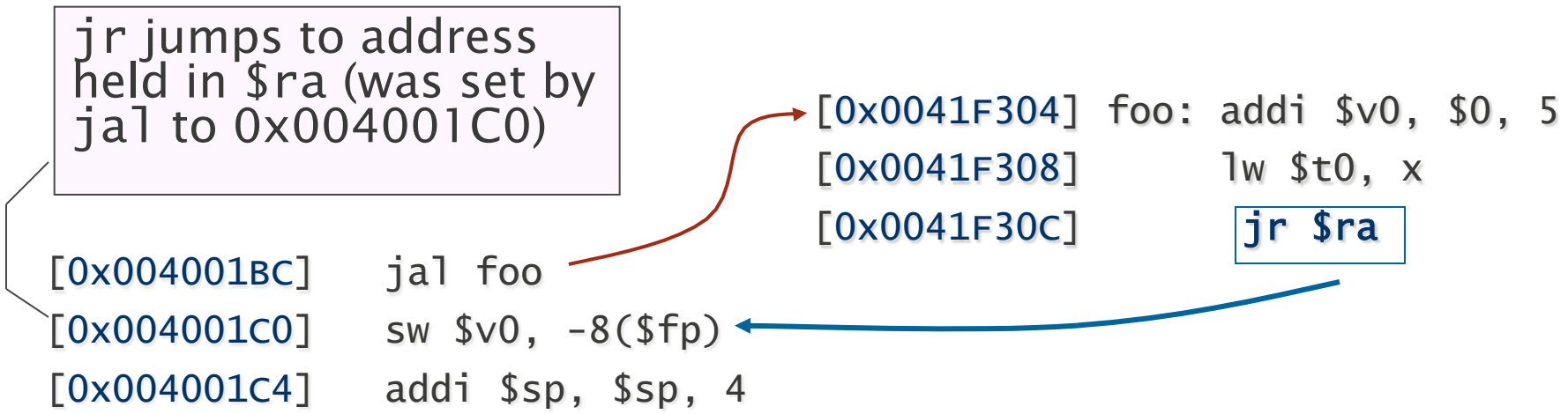
Function calling in MIPS

- **Remember: `jal` instruction for calling a function**
  - `jal` is like jump (`j`), but it first saves the address of the instruction following the `jal` (ie. at address =PC+4) in register `$ra`



Function calling in MIPS

- **Remember: `jr` instruction for returning from a function**
  - `jr` jumps to the absolute address held in a specified GPR register (usually `$ra`)



Function calling in MIPS

- **To write a function**
  - Put label at the start of the function
  - Write body of the function
  - End function with `jr $ra`
- **To call a function**
  - Write `jal label`
  - When the function returns, program will continue from the next instruction
- **Is this really all it is needed to call a function?**

Passing data

- **Some functions have parameters**
  - Need a way of passing values for these parameters (i.e., arguments) from caller to function
- **Some functions return values**
  - Need a way of getting the return value safely back to caller
- **Solution: reserve some registers for these tasks**
  - Can extend the "syscall" data passing method:
  - Pass function parameters in `$a0`, `$a1`, `$a2`, `$a3`
  - Return values in `$v0`, `$v1`
- **This convention still has some problems**
  - Will make a better system later
  - But let's see how it works for now

Focus on the blue parts

Example

```
# Part of program that uses
# function fact

        ...
        # Read int from user
        addi  $v0, $0, 5
        syscall

        # Pass parameter in $a0
        addi  $a0, $v0, 0
        # Call!
        jal  fact
        # Put result in $t1
        addi  $t1, $v0, 0

        # Print factorial
        addi  $v0, $0, 1
        addi  $a0, $t1, 0
        syscall

        # Exit
        addi  $v0, $0, 10
        syscall
```

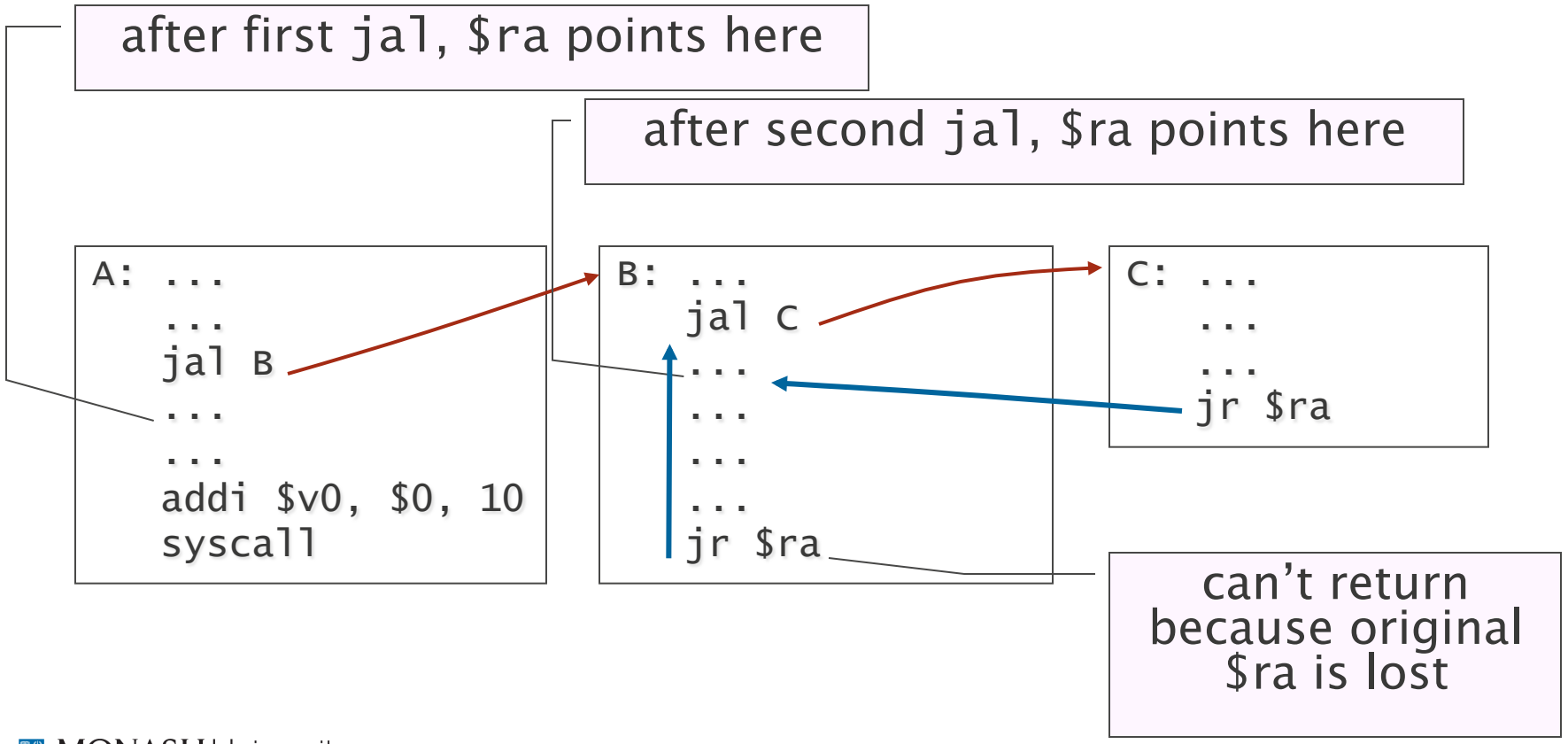
```
# Factorial function

fact:   addi  $t0, $0, 1  # result=1

        # Repeat while parameter
        # in $a0 is > 1
loop:   addi $t1, $0, 1
        slt $t1, $t1, $a0
        beq $t1, $0, end

        # Multiply result by
        # parameter
        mult  $t0, $a0
        mflo $t0

        # Decrement parameter
        addi  $a0, $a0, -1

        # loop back to test
        j  loop

        # At end, $t0 contains
        # factorial of user input
end:    addi $v0, $t0, 0 #$v0=answer
        jr  $ra           # Return
```

Limitations

- **This simple function-calling convention works, but has limits**
- **Function must:**
  - Not call other functions (i.e., it is a “leaf function”)
  - Not use more than four parameters ($a0-$a3)
  - Only write to “safe” registers $v0-$v1, $a0-$a3, $t0-$t9
  - Not have/use local variables
- **For more sophisticated function calling, one needs more sophisticated convention**

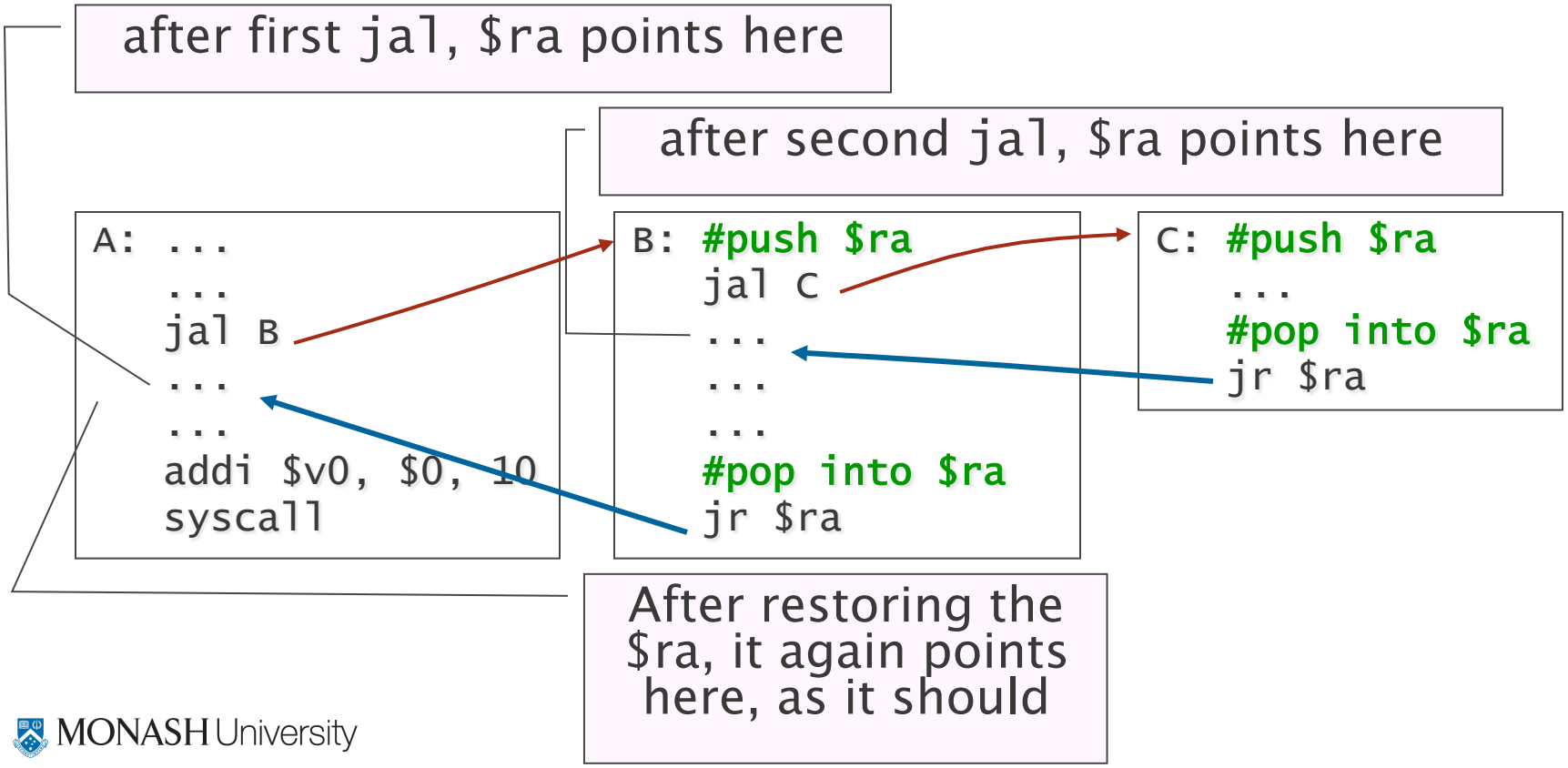
Limitation: calling other functions

- **Return address register (`$ra`) can hold only one value**



Limitation: calling other functions

- **Return address register ($ra) can hold only one value**
- **Solution: save and restore $ra register on the stack upon function entry/exit (push on entry, pop on exit)**



Limitation: passing arguments

- **Not enough registers (4) to pass arguments to some functions**

```
addi $a0, $0, 1
lw $a1, x
addi $a2, $0, 0
lw $a3, -4($fp)
addi $??, $0, 2
jal five
...
```

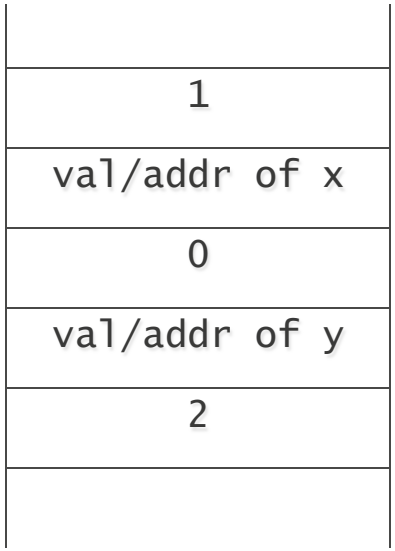
no such register $a4

```
five: # takes 5
      # parameters
      ...
      # examine
      # $a0, etc
      ...
      jr $ra
```

Limitation: passing arguments

- **Not enough registers (4) to pass arguments to some functions**
- **Solution: pass arguments on stack**

```
# push 2
# push global y
# push 0
# push local x
# push 1
jal five
# pop
# pop
# pop
# pop
# pop
```




```
five: # takes 5
      # parameters
      ...
      # examine
      # stack
      ...
      jr $ra
```

For simplicity, in FIT2085 we will pass all arguments onto the stack

Limitation: saving registers

- Function may use registers which hold important values

```
...
lw $t0, a
...
...
jal func
...
...
# $t0 has been
# changed!
add $t0, $t0, $v0
...
```

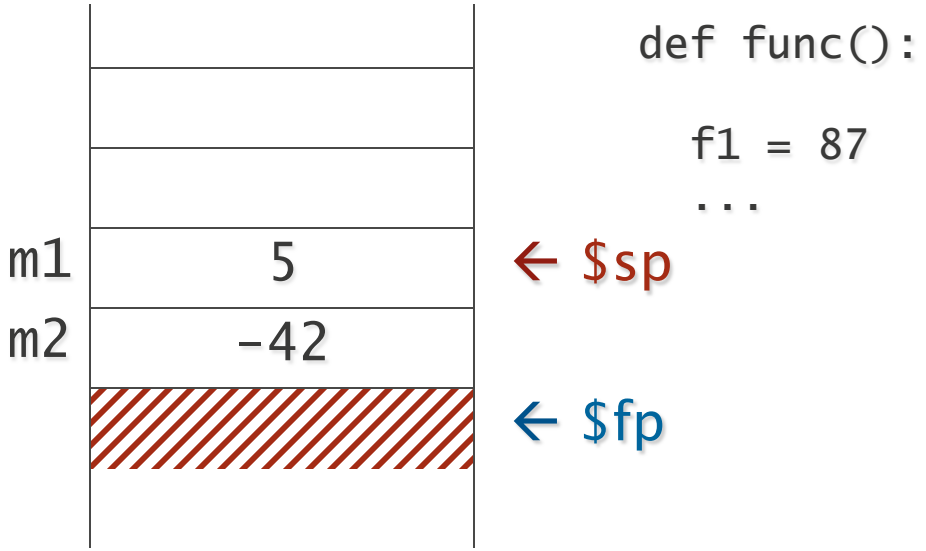
```
func: ...
      # trashes
      # $t0
      lw $t0, x
      ...
      jr $ra
```

- Solution: save/restore registers on stack (but not in FIT2085)
- In FIT2085 you are not allowed to do this; assemblers are and do!

Limitation: Local variables (reminder)

- **A function needs its own local variable storage**

```
def main():

  m1 = 5
  m2 = -42
   ...
  func()
```

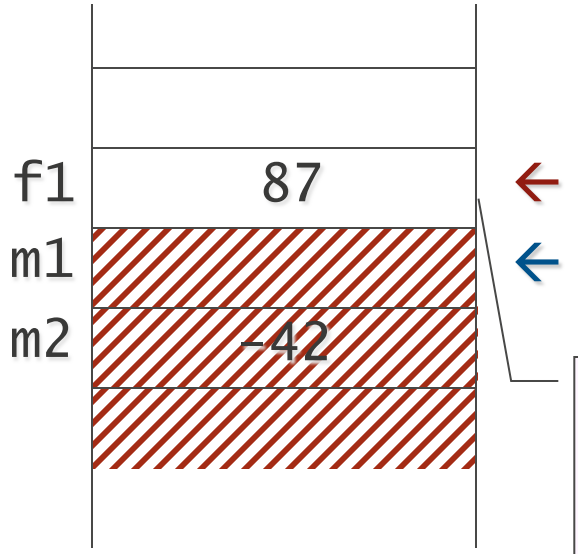


Limitation: Local variables (reminder)

- A function needs its own local variable storage
- Solution: function adjusts $fp to top of stack, then allocates its own locals

```
def main():

  m1 = 5
  m2 = -42
   ...
  func()
```

inside func, m1 and m2 are inaccessible



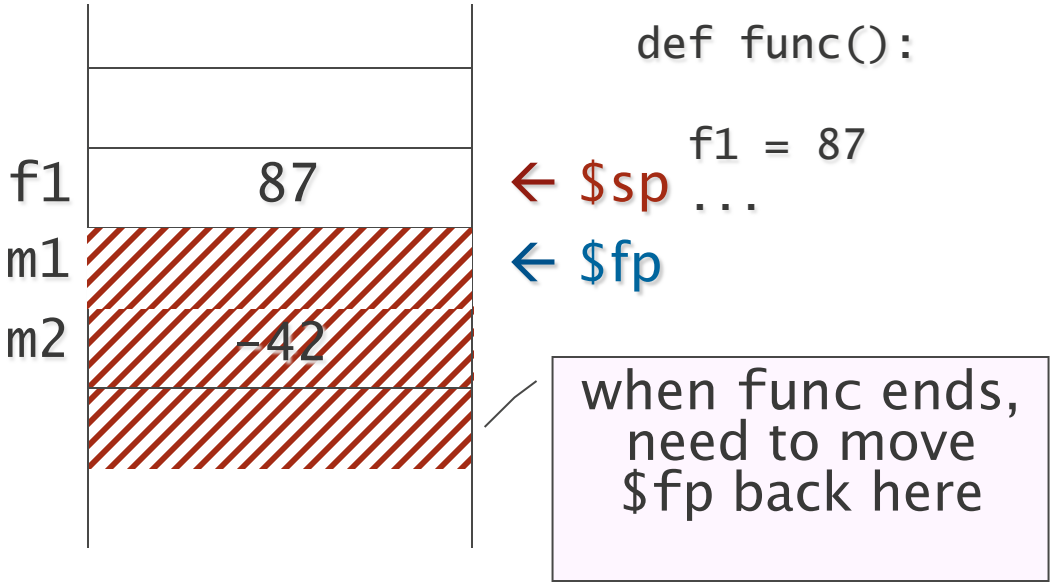

```
def func():

  f1 = 87
  ...
```

inside func, f1 is accessible at -4($fp)

Limitation: Local variables – restore $fp

- **On function return, stack state (including `$fp`) must be restored**

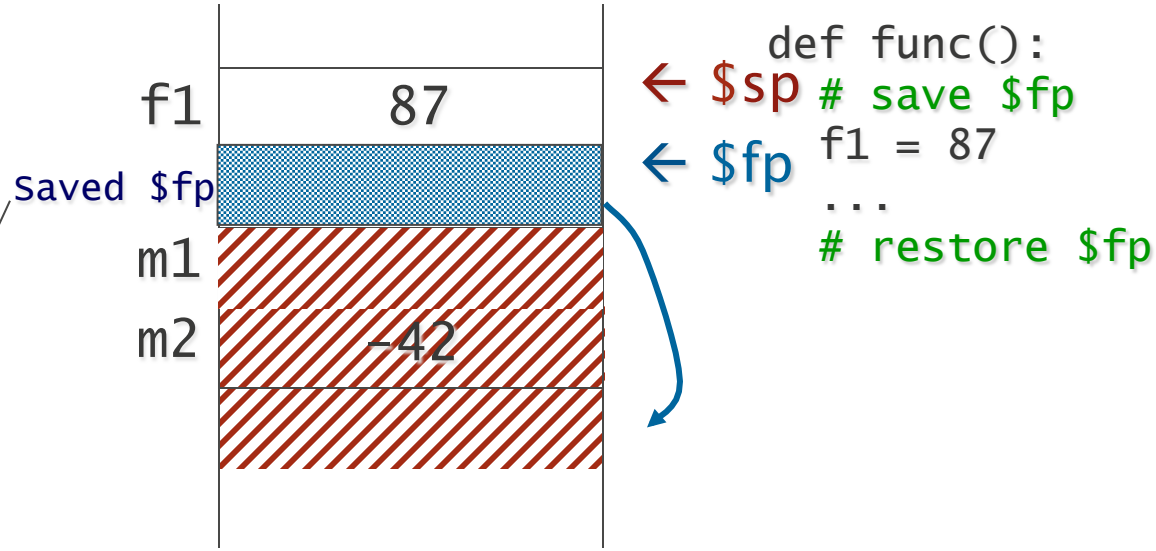
```
def main():

  m1 = 5
  m2 = -42
   ...
  func()
```



Limitation: Local variables – restore $fp

- On function return, stack state (including `$fp`) must be restored
- Solution: save/restore `$fp` on stack

```
def main():

  m1 = 5
  m2 = -42
   ...
  func()
```



by saving old `$fp`, it can be restored at function's end

Function calling convention

Task	Whose responsibility?
save / restore temporary registers	caller
pass / clear arguments	caller
save / restore `$ra`	callee
save / restore `$fp`	callee
allocate / deallocate local variables	callee

Function calling convention

When calling a function, caller:

1. Saves temporary registers by pushing their values on stack
2. Pushes arguments on stack
3. Calls the function with `jal` instruction

On function entry, callee:

1. Saves `$ra` by pushing its value on stack
2. Saves `$fp` by pushing its value on stack
3. Copies `$sp` to `$fp`
4. Allocates local variables

Example: caller

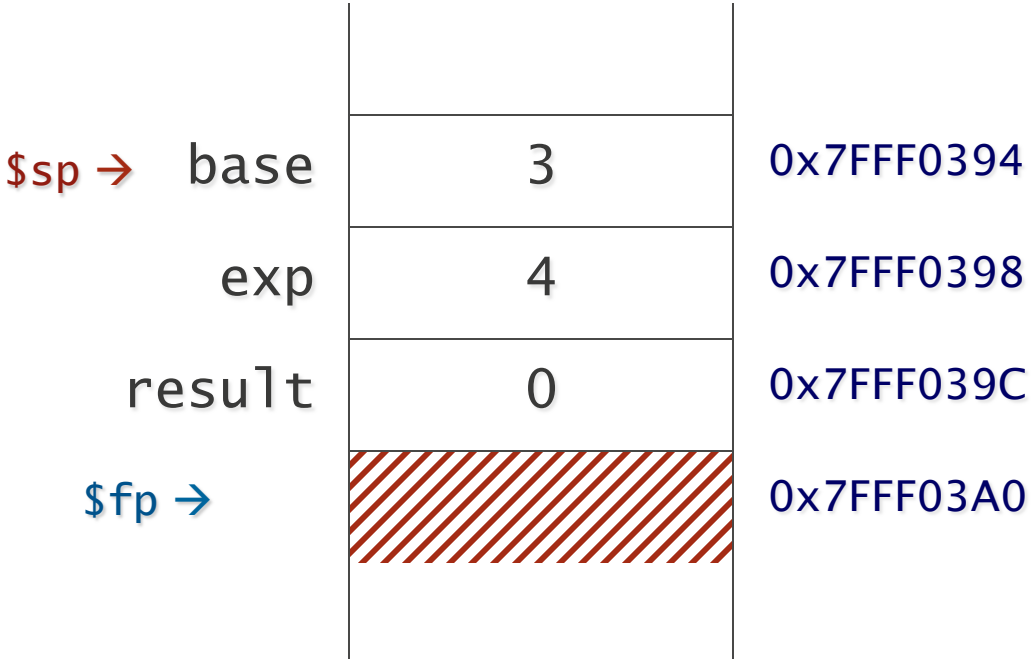
```
# Python program which calls a function.

def power(b, e):
    ...
    return result

def main():
    base, exp, result = 0, 0, 0
    read(base)
    read(exp)

    result = power(base, exp)

    print(result)
```

This is the memory diagram at this point

base is at -12($fp)

exp is at -8($fp)

result is at -4($fp)



Assume user has entered 3 for base and 4 for exp

Example: caller

```
# Python program which calls a function.

def power(b, e):
    ...
    return result

def main():

    base, exp, result = 0, 0 ,0
    read(base)
    read(exp)

    result = power(base, exp)

    print(result)
```

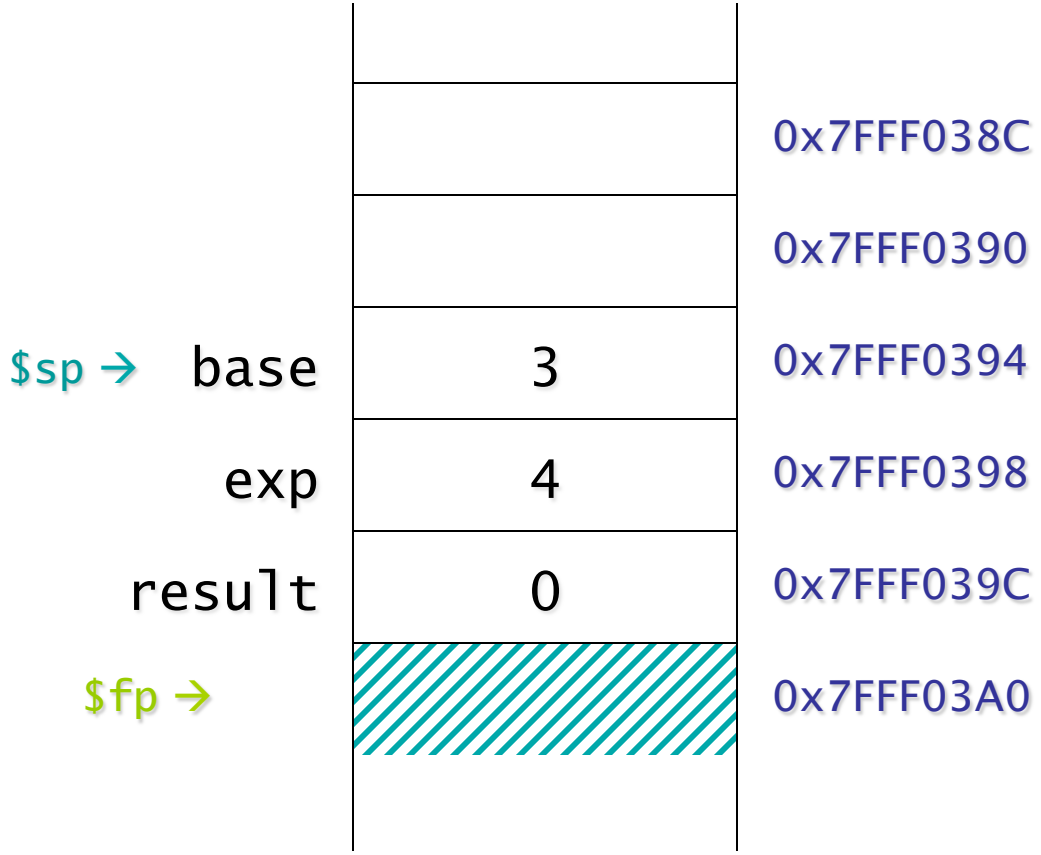
```
        .text
main:   # 3 * 4 = 12 bytes local
        addi $fp, $sp, 0
        addi $sp, $sp, -12

        # Initialize locals
        sw $0, -12($fp)
        sw $0, -8($fp)
        sw $0, -4(4fp)

        addi $v0, $0, 5
        syscall
        sw $v0, -12($fp)  # base

        addi $v0, $0, 5
        syscall
        sw $v0, -8($fp)   # exp

# Now we are up to the
# function call ...
```

base is at -12($fp)
exp is at -8($fp)
result is at -4($fp)

Example: caller

caller step 1: save temporary registers by pushing their values on stack

not needed in FIT2085

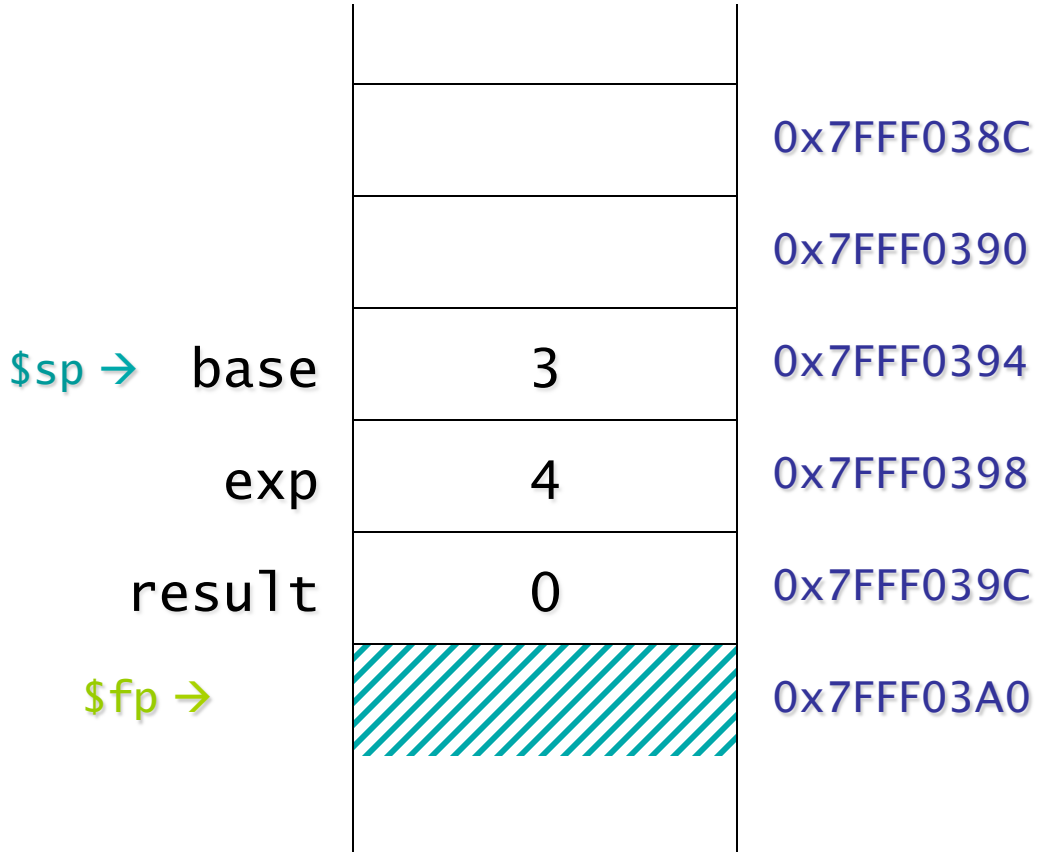


Example: caller

caller step 2: push function arguments onto the stack

two arguments, called b and e

```
def power(b, e):
```



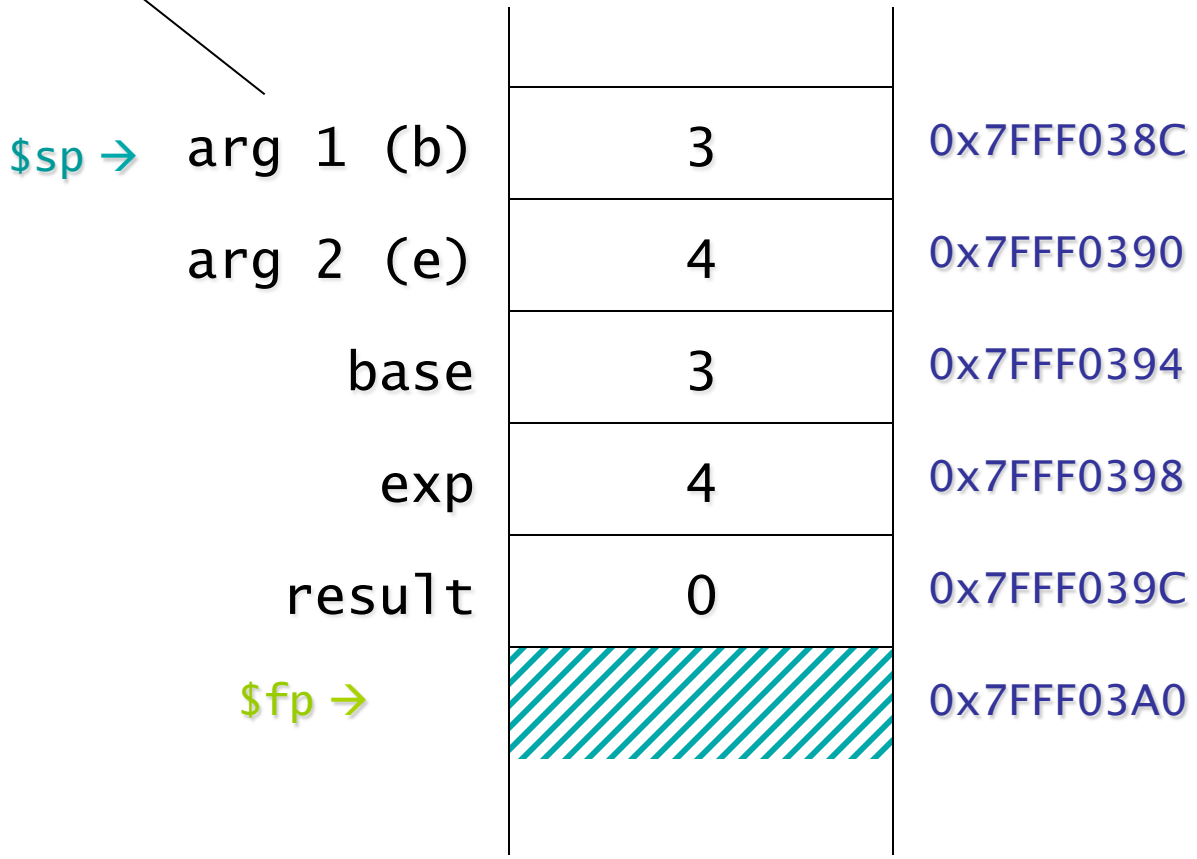
Example: caller

note offsets of arguments:
b at 0($sp)
e at 4($sp)

caller step 2: push function arguments onto the stack

two arguments, called b and e

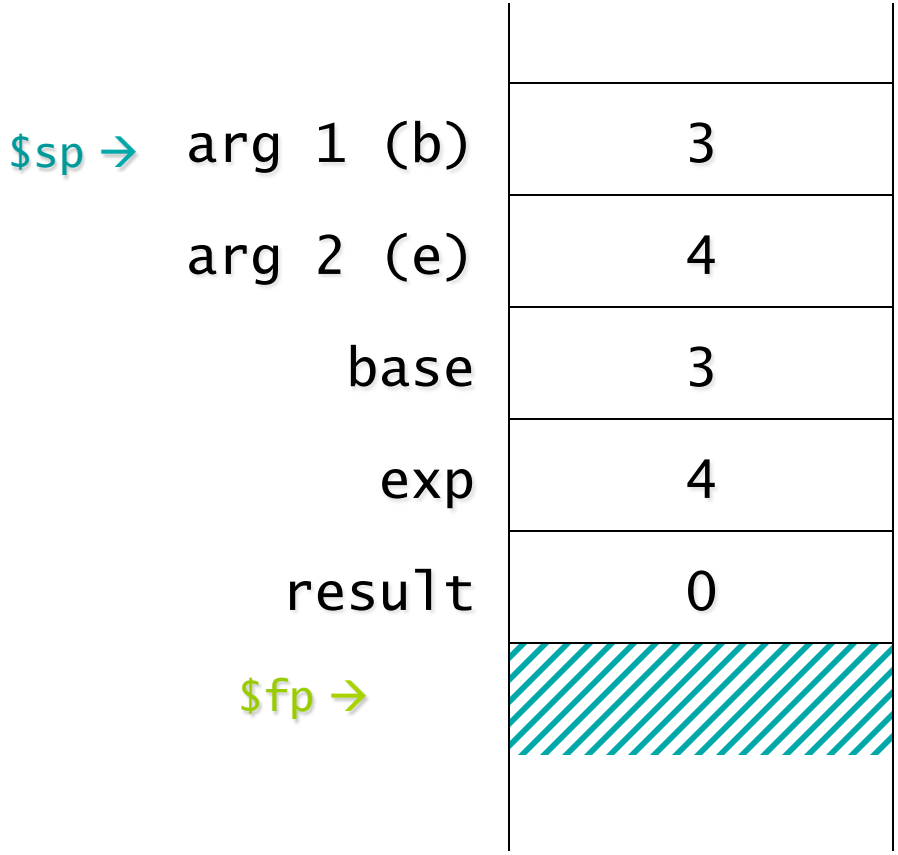
```
def power(b, e):
```



Example: caller

caller step 3: call function with `jal` instruction

no effect visible on the stack



Example: caller

```
# Python program which calls a function.

def power(b, e):
    ...
    return result

def main():
    base, exp, result = 0, 0 ,0
    read(base)
    read(exp)

    result = power(base, exp)

    print(result)
```

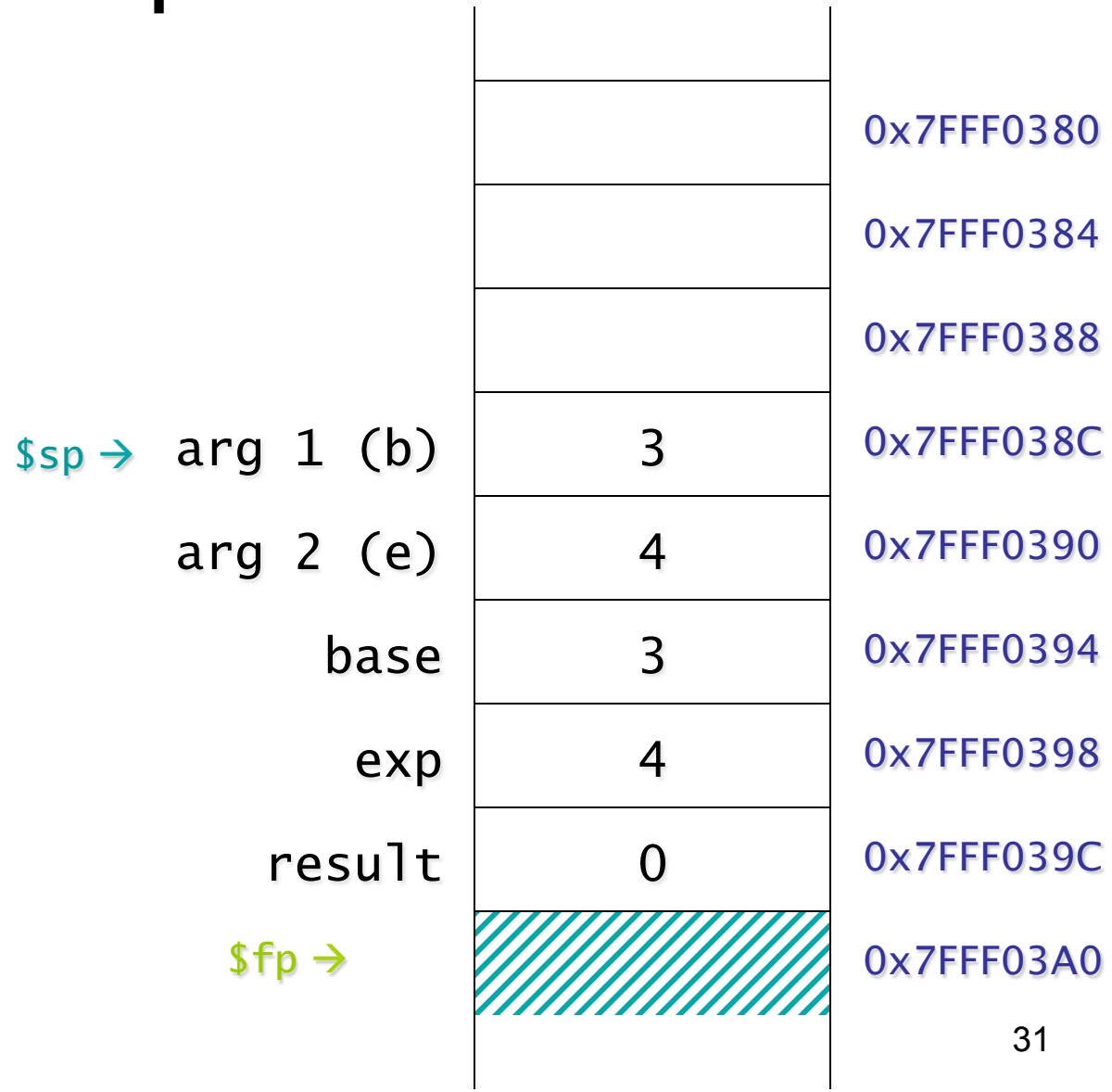
base is at -12($fp)
exp is at -8($fp)
result is at -4($fp)
arg1 at 0($Sp)
arg2 at 4($sp)

```
# ... continued from
# previous slide

    # Call function.
    # push 2 * 4 = 8 bytes
    # of arguments
    addi $sp, $sp, -8

    # arg 1 = base
    lw $t0, -12($fp) # base
    sw $t0, 0($sp)   # arg 1

    # arg 2 = exp
    lw $t0, -8($fp)  # exp
    sw $t0, 4($sp)   # arg 2

    # Call power function
    jal power


# main function To Be
# Continued next lecture ...
```

Example: callee

```
# A function that gets called.

def power(b, e):

    result = 1

    # Keep going while
    # exponent is positive.

    while e > 0:

        # Multiply by base.
        result *= b

        # Less to multiply.
        e -= 1

    # Return the result to the caller.
    return result
```

Example: callee



callee steps 1 and 2: save $ra and $fp by pushing their values on stack

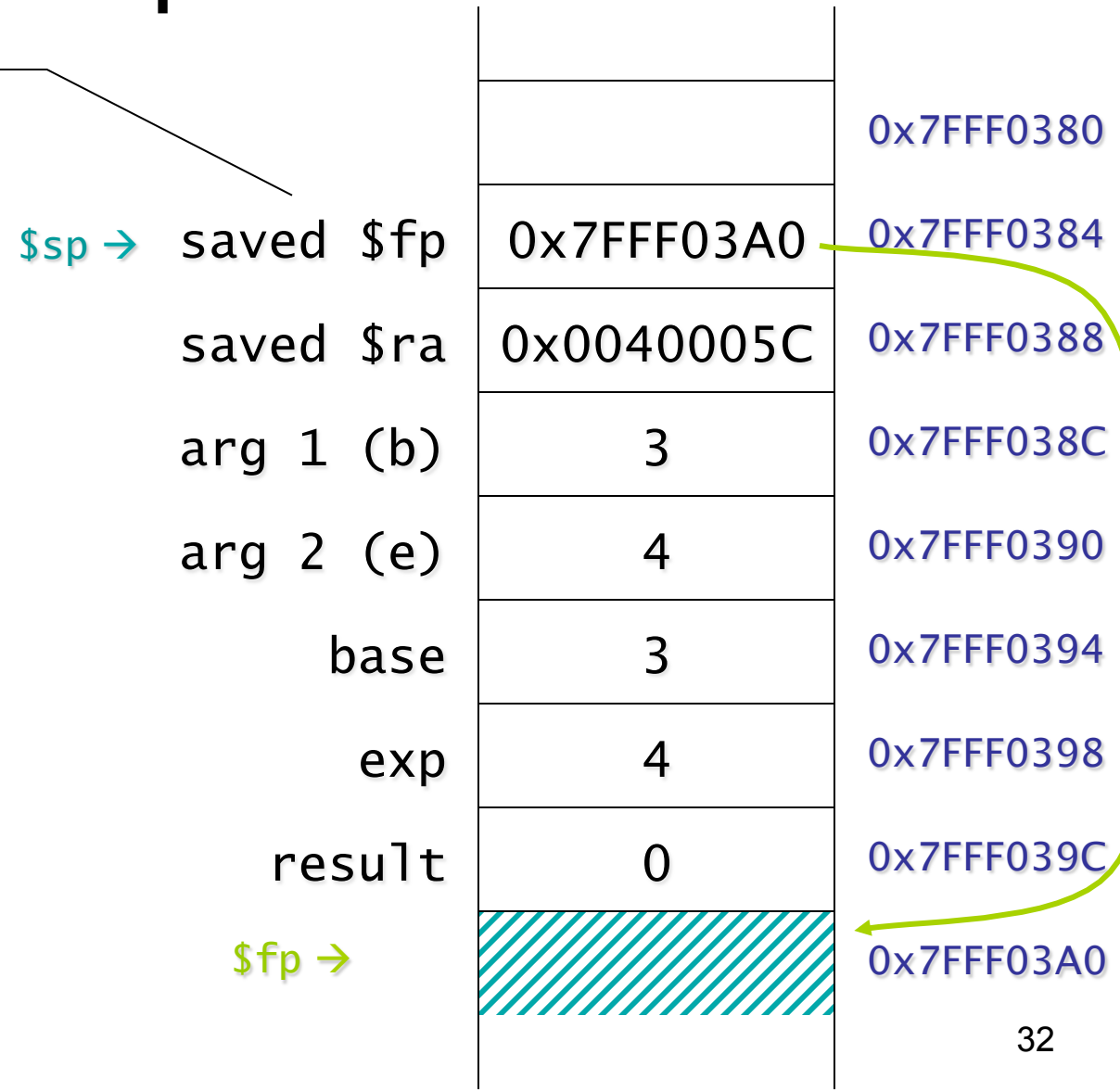
can do both these steps at once

Example: callee

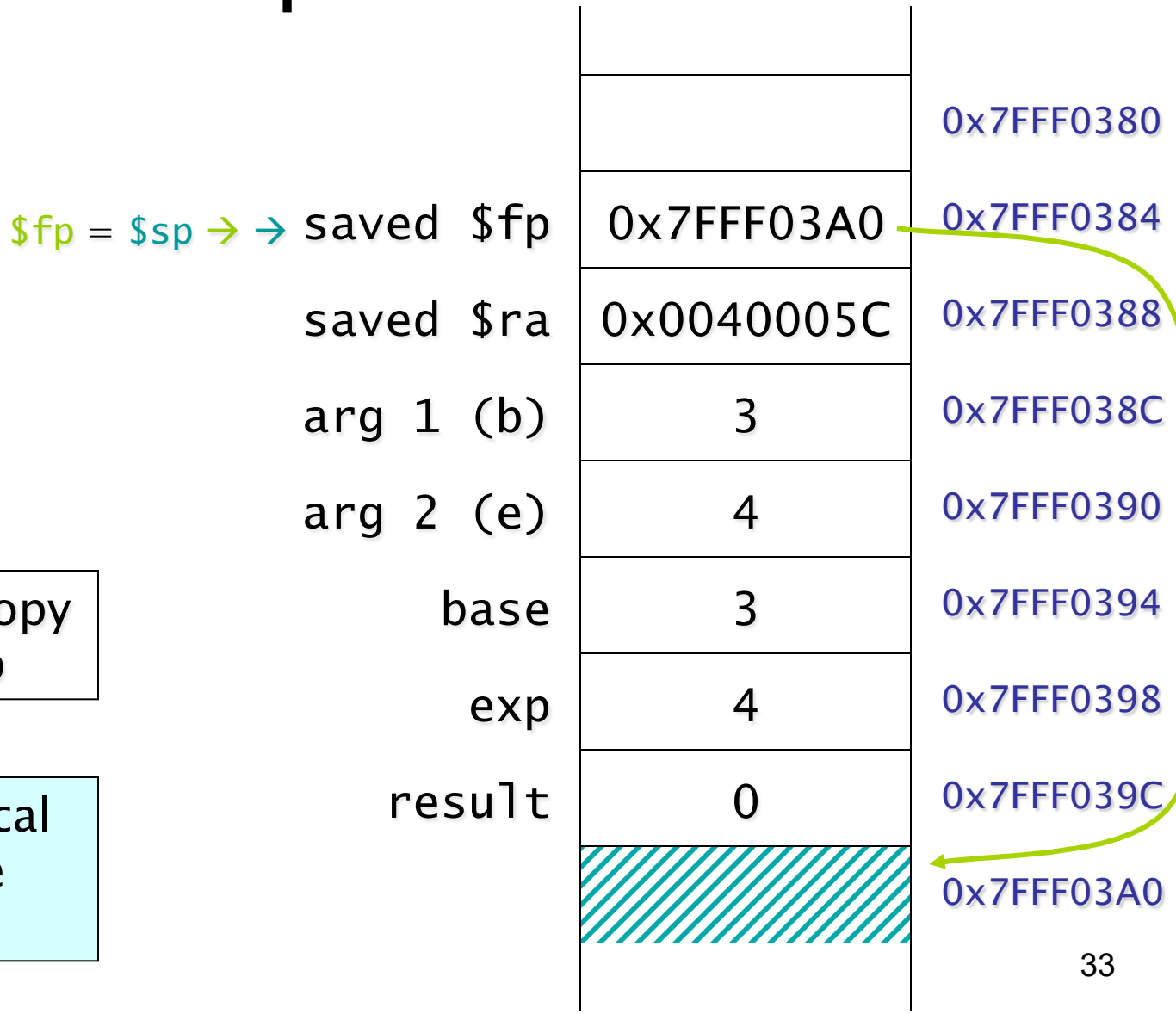
saved $fp contains address of another location on stack

callee steps 1 and 2: save $ra and $fp by pushing their values on stack

can do both these steps at once



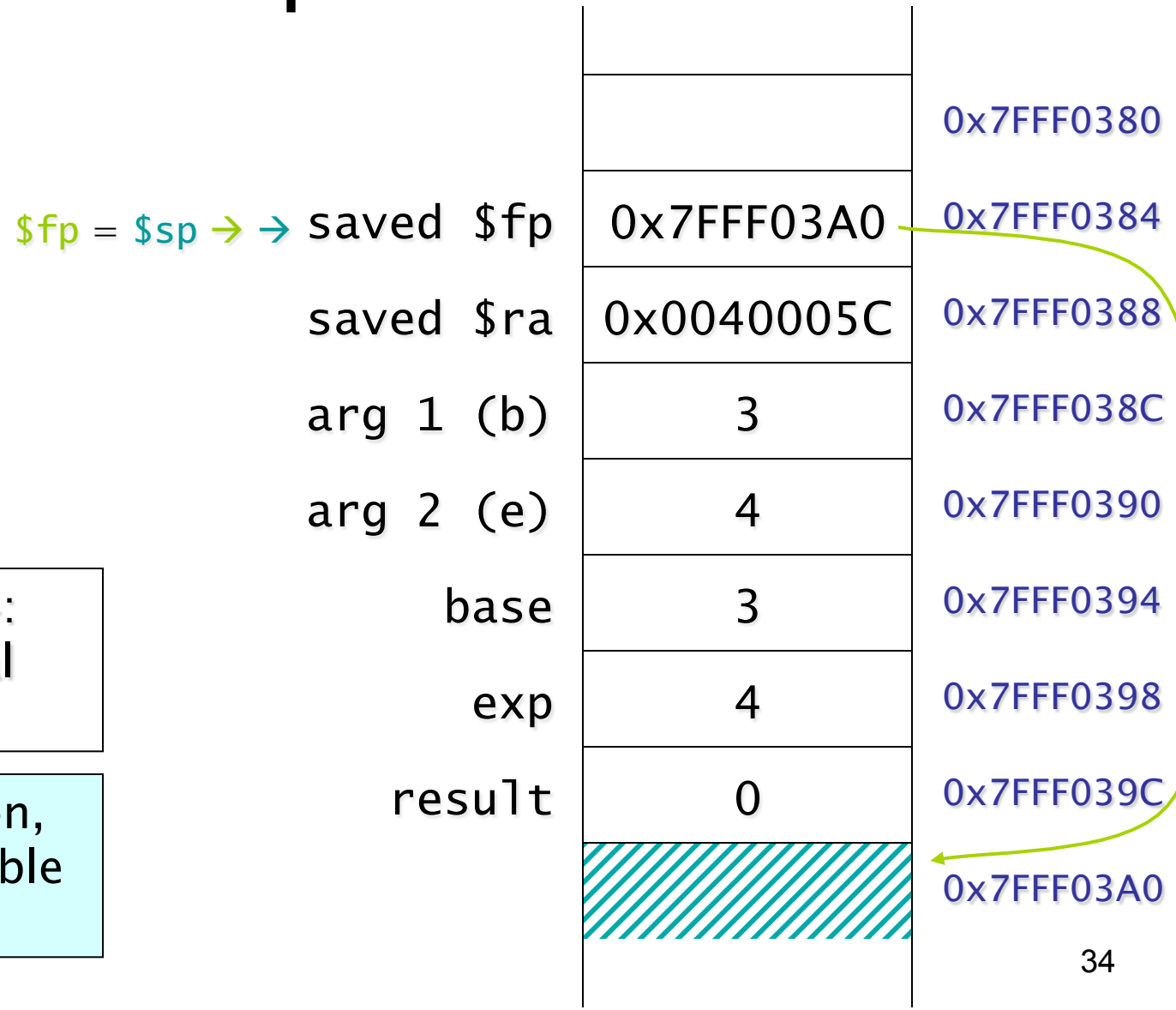
Example: callee



callee step 3: copy $sp into $fp

now main's local variables are inaccessible

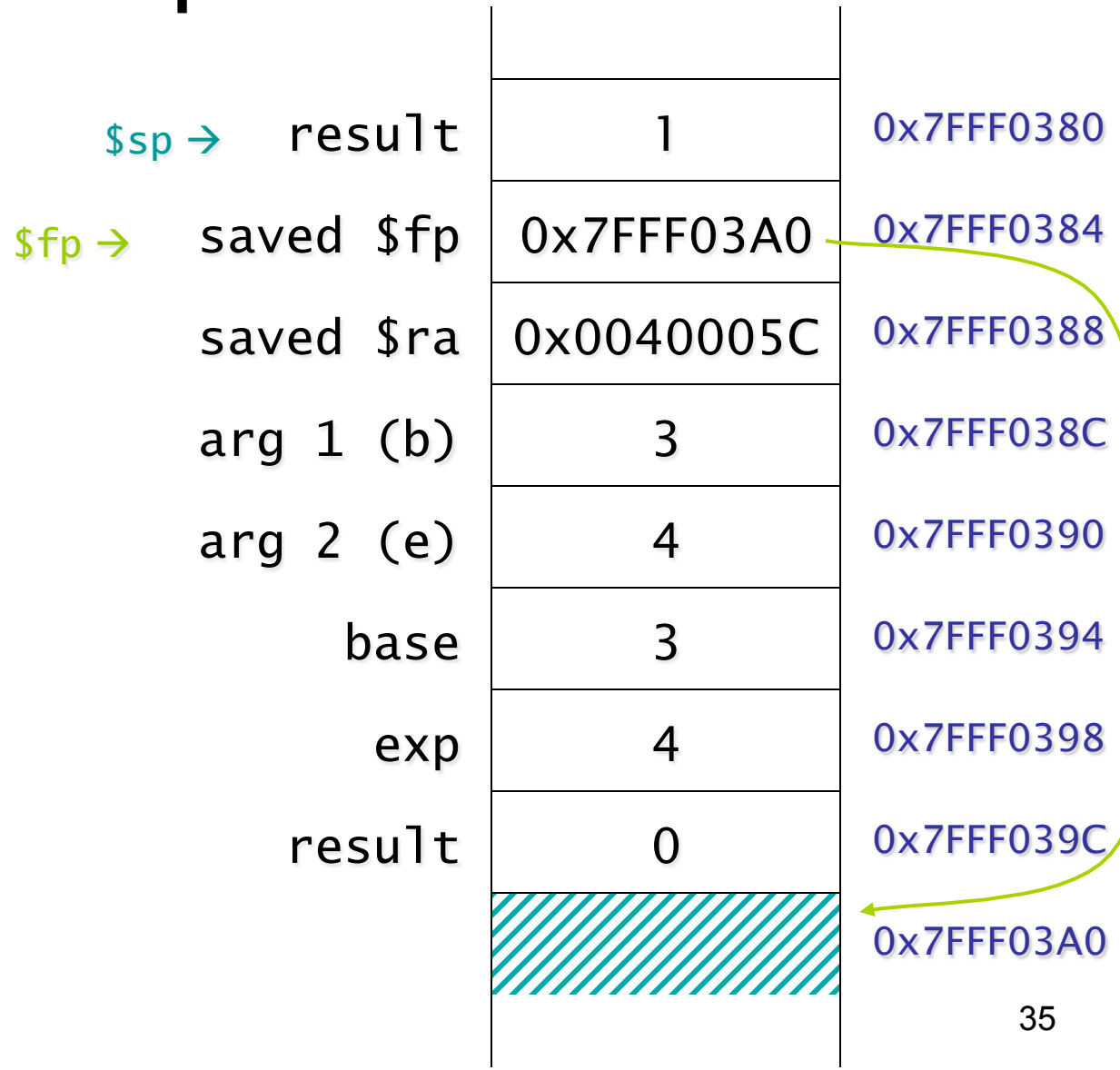
Example: callee



callee step 4: allocate local variables

in this function, one local variable (`result`)

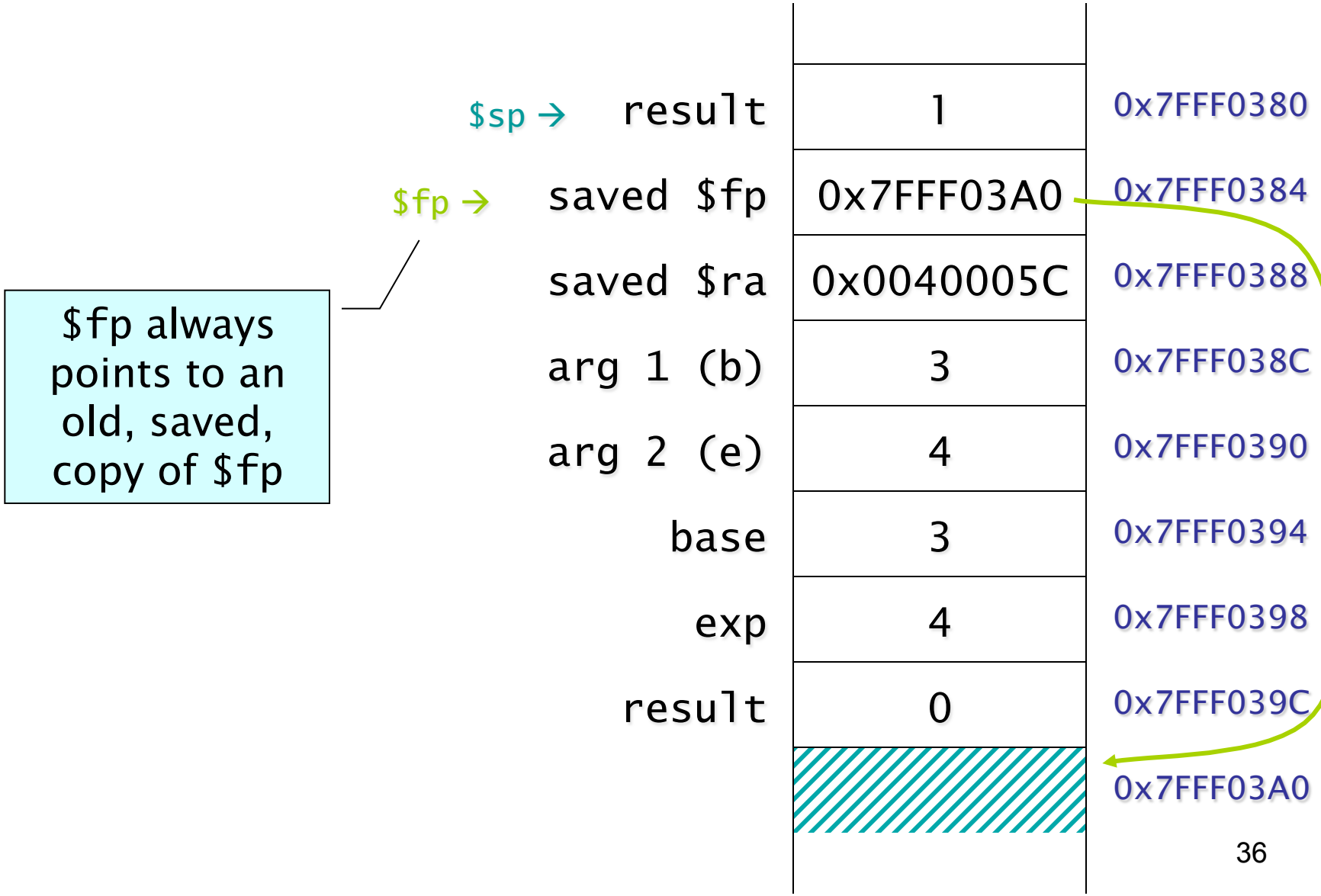
Example: callee



callee step 4: allocate local variables

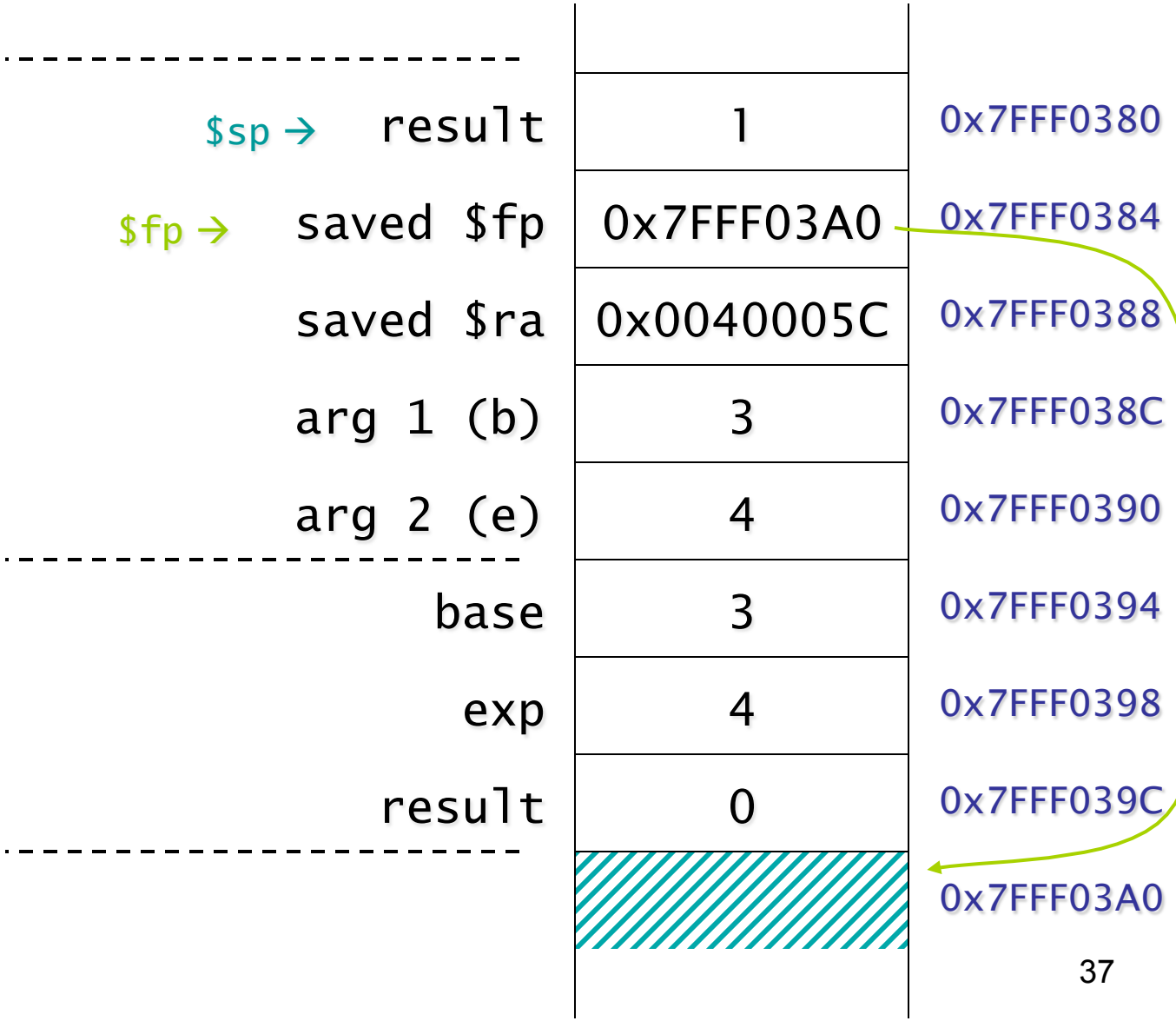
in this function, one local variable (`result`)

Frame Pointers

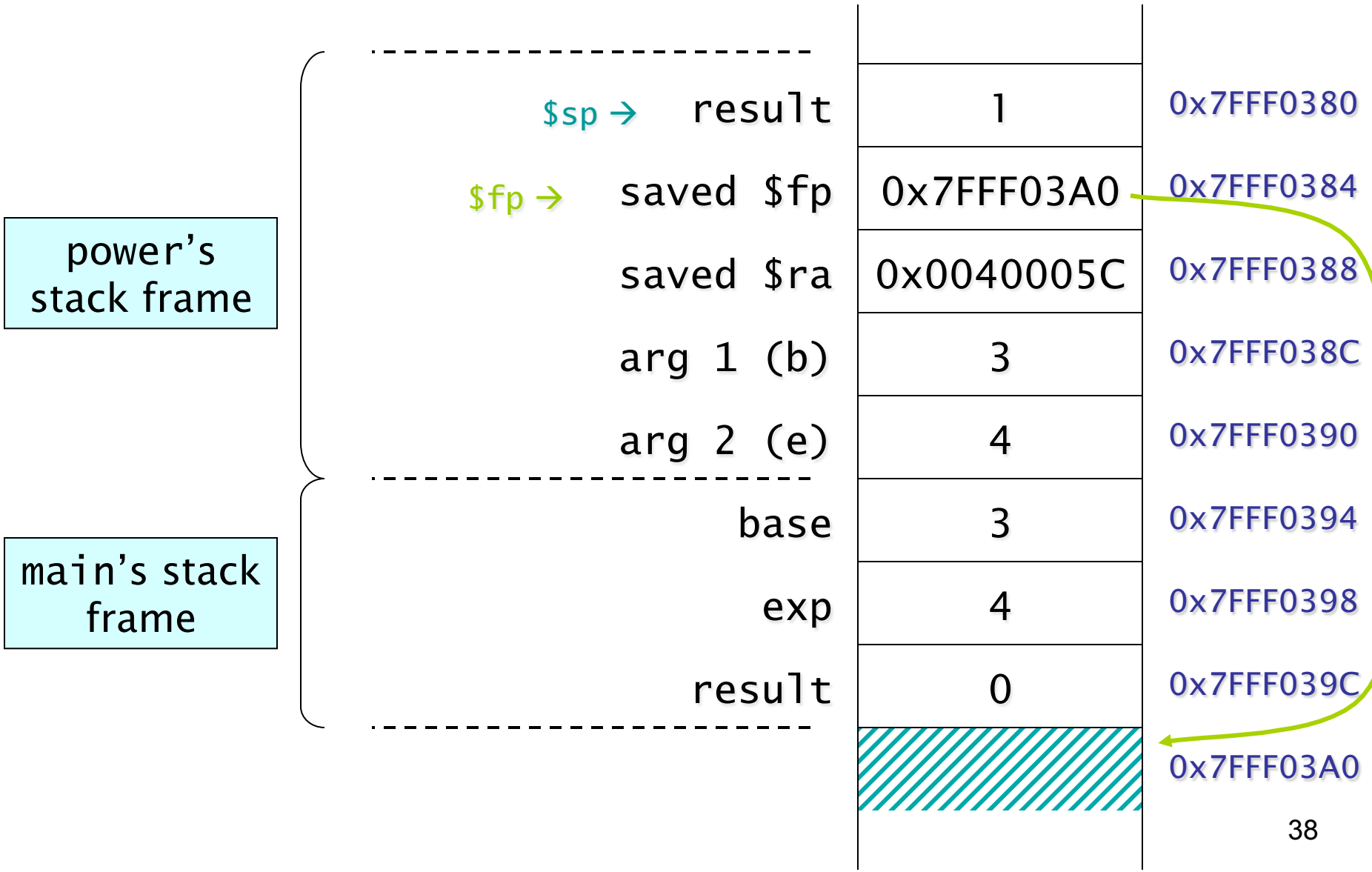


Stack Frames

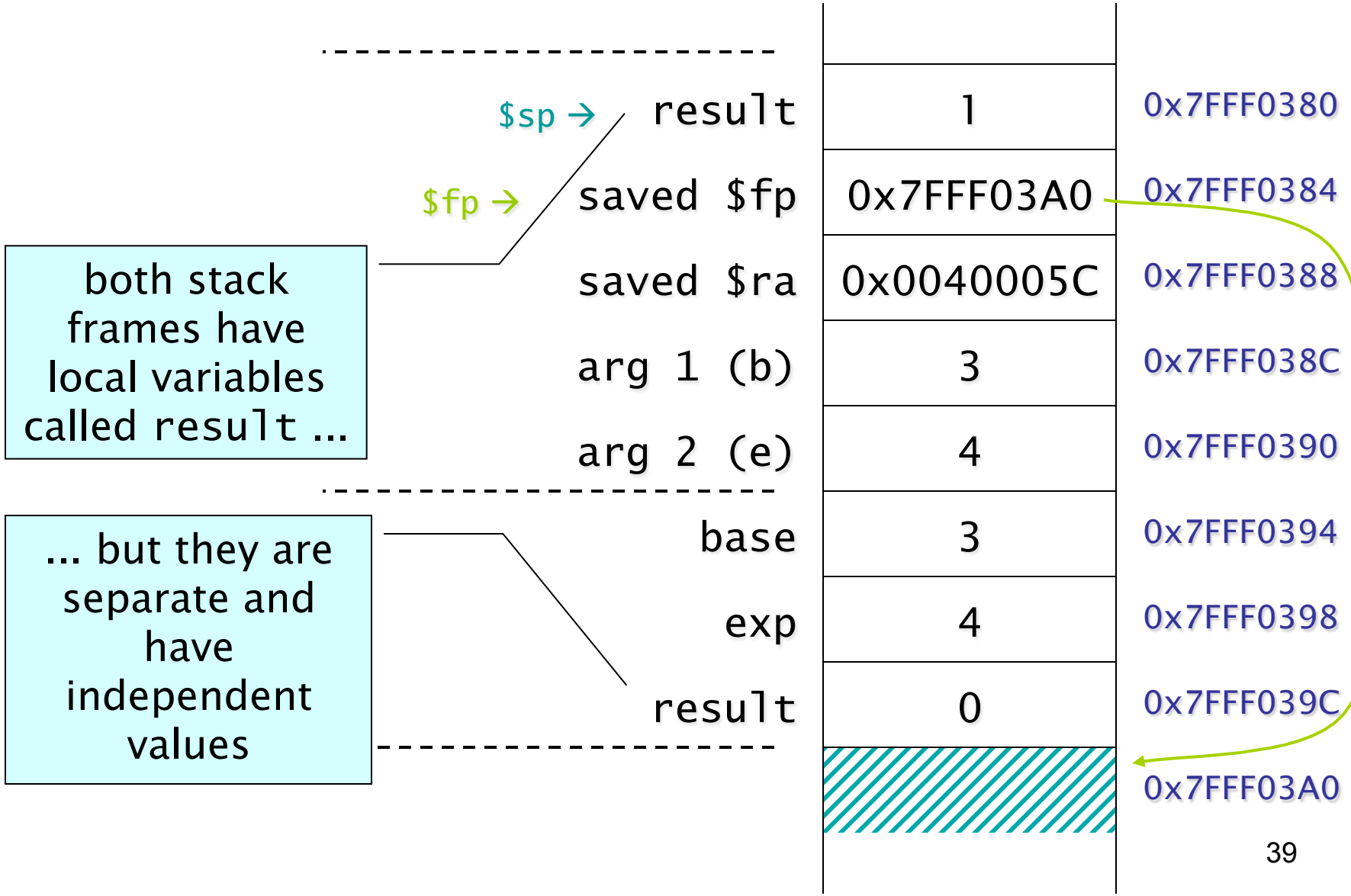
the data on the stack that is associated with a function is called a stack frame



Stack Frames

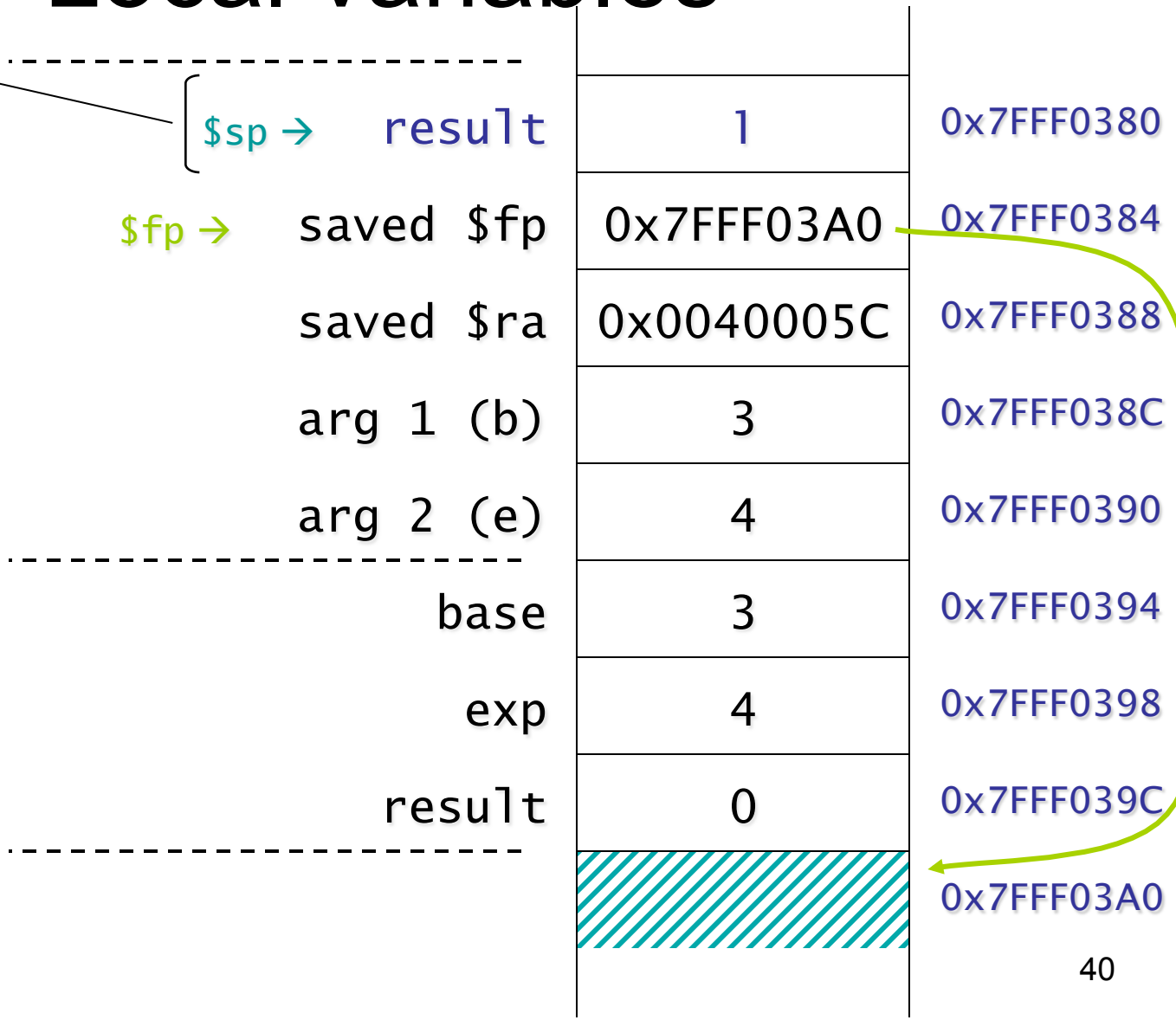


Stack Frames



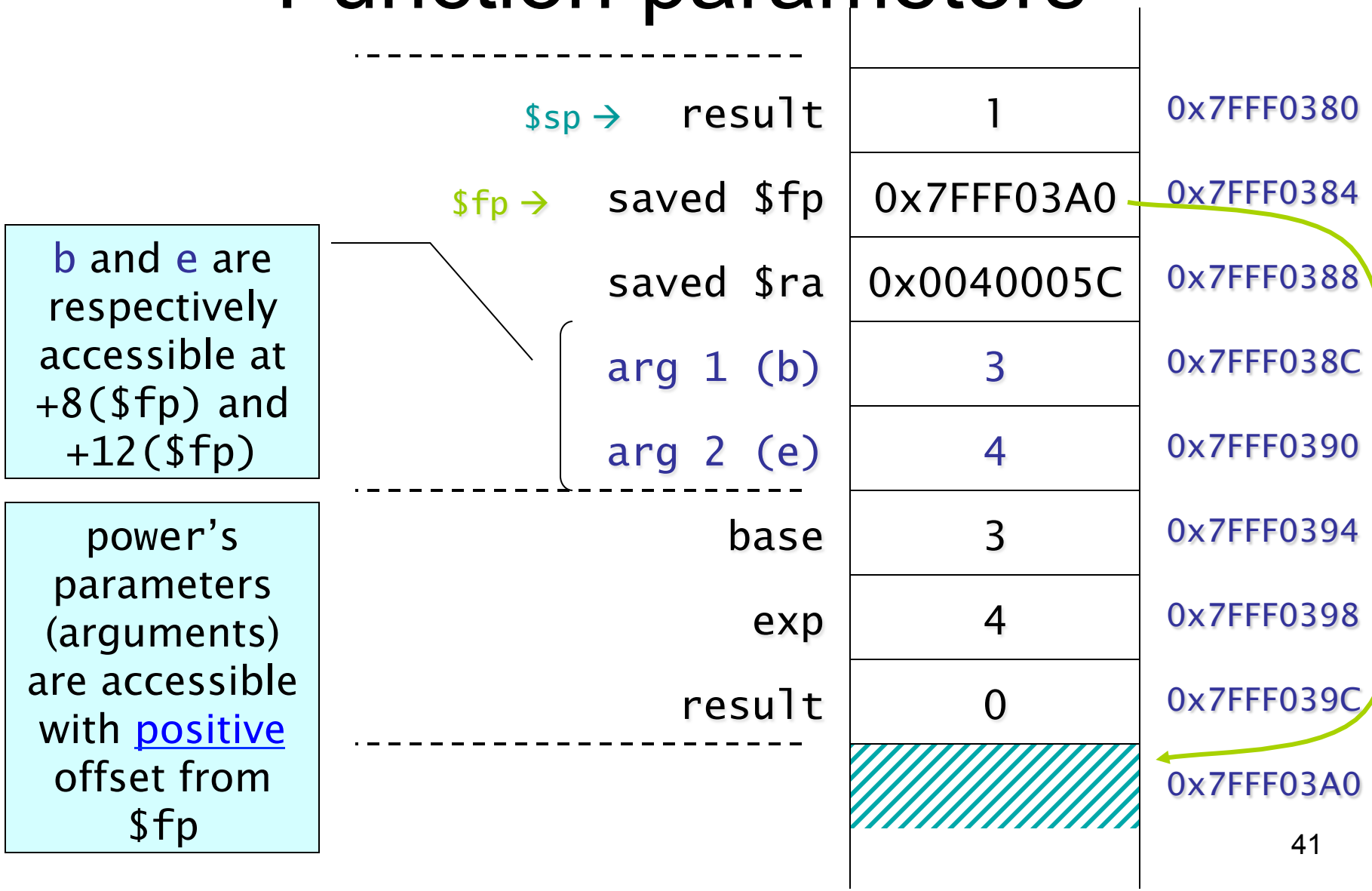
Local variables

result is at -4($fp)



power's local variables are accessed as main's were, with negative offsets from $fp

Function parameters



Example: callee

```
# A function that gets called.

def power(b, e):
    result = 1

    # Keep going while
    # exponent is positive.

    while e > 0:

        # Multiply by base.
        result *= b

        # Less to multiply.
        e -= 1

    # Return the result to the caller.
    return result
```

```
power:  # Save $ra and $fp
        addi $sp, $sp, -8
        sw $ra, 4($sp)
        sw $fp, 0($sp)

        # Copy $sp to $fp
        addi $fp, $sp, 0

        # Alloc local variables
        # 1 * 4 = 4 bytes.
        addi $sp, $sp, -4

        # Initialize locals.
        addi $t0, $0, 1
        sw $t0, -4($fp)  # result

        # Now we are inside
        # the function body.
```

result is at -4($fp)

b is at 8($fp)

e is at 12($fp)

Example: callee

```
# A function that gets called.

def power(b, e):

    result = 1

    # Keep going while
    # exponent is positive.

    while e > 0 :

        # Multiply by base.
        result *= b

        # Less to multiply.
        e -= 1

    # Return the result to the caller.
    return result
```

```
loop:   # Stop if !(e > 0)
        lw $t0, 12($fp)  # e
        slt $t0, $0, $t0
        beq $t0, 0, end

        # result = result * b
        lw $t0, -4($fp)  # result
        lw $t1, 8($fp)   # b
        mult $t0, $t1
        mflo $t0
        sw $t0, -4($fp)  # result

        # e = e -1
        lw $t0, 12($fp)  # e
        addi $t0, $t0, -1
        sw $t0, 12($fp)  # e

        # Repeat loop.
        j loop

end:    # Now ready to return.
# Continued in next lecture ...
```

result is at -4($fp)

b is at 8($fp)

e is at 12($fp)

Summary

- **Function calling**
  - `jal` and `jr` instructions
- **Function calling convention**
  - For making a function call
- **Structure of stack**
  - stack frames